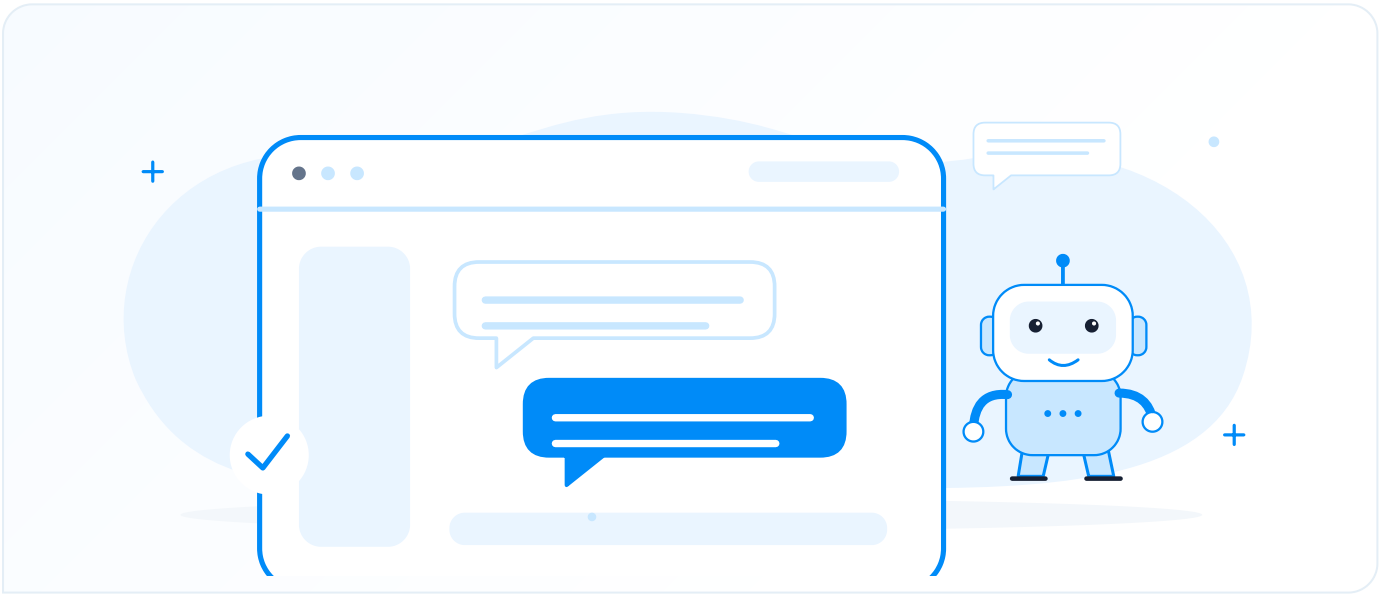
Tutoriel GenAI · Ollama · Streamlit · Déploiement local

Déployer un assistant GenAI local avec Streamlit

Construire un assistant de questions-réponses réellement local avec Streamlit et Ollama, puis conserver un mode mock pour les tests rapides.

Auteur **Youness Boutyour**



DURÉE	NIVEAU	FORMAT
50 minutes	Débutant avancé	Tutoriel pratique

GenAI · Ollama · Streamlit · Déploiement local · publié le 2026-08-21 · local

Objectifs et livrables

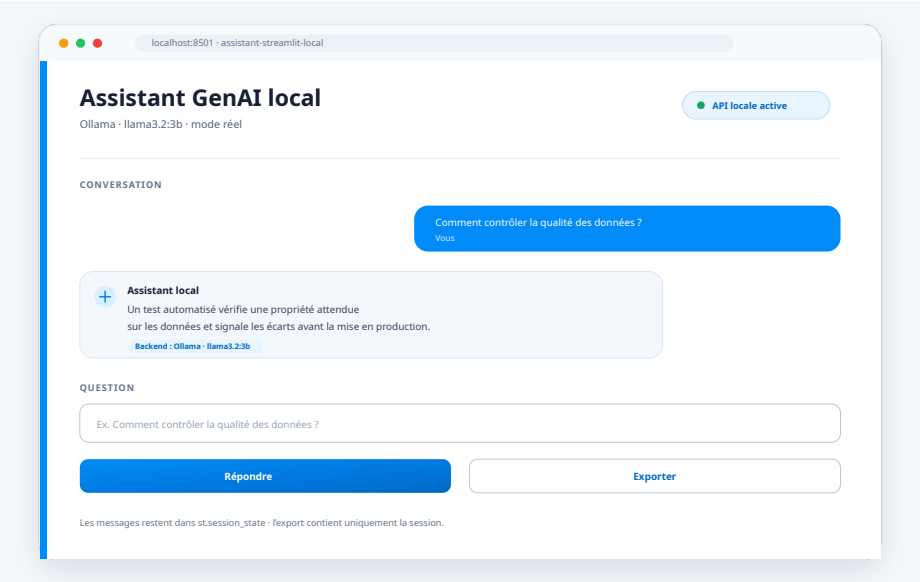
Pour faire essayer un assistant à quelques collègues sans envoyer les questions vers un service distant, nous allons construire une interface Streamlit reliée à Ollama. Ollama exécute le modèle sur la machine et expose une API HTTP locale ; l'application ne change donc pas de contrat lorsqu'elle passe du mode de test à une vraie génération.

Le parcours commence par l'installation d'Ollama et le téléchargement de `llama3.2:3b`, un modèle suffisamment compact pour une première démonstration. Nous vérifierons ensuite l'endpoint `/v1/chat/completions`, configurerons le modèle par variables d'environnement et lancerons l'interface en mode réel. Le mode mock restera disponible pour isoler les problèmes d'interface d'un problème de modèle ou de mémoire.

Vous verrez comment distinguer une réponse générée par Ollama d'une source documentaire, comment diagnostiquer un modèle absent ou un serveur arrêté, et comment conserver l'historique dans `st.session_state`. La configuration reste côté processus Streamlit et l'export ne contient que les messages de la session, jamais une clé ou un paramètre d'accès.

À la fin, vous disposerez d'un assistant local fonctionnel, d'un bouton d'export vers `outputs/session.json`, d'un contrôle de non-divulgation et d'un mode mock de secours. Vous pourrez remplacer `llama3.2:3b` par un autre modèle installé dans Ollama sans réécrire l'interface.

Le flux réel reste local : Streamlit appelle l'API Ollama avant d'exporter la session.



Aperçu de l'interface utilisée : saisie d'une question, réponse du modèle local et export de session.

Lire le flux avant de coder

Le schéma sépare ce que voit l'utilisateur de ce qui reste côté serveur. Streamlit collecte la question, envoie un message à Ollama sur `127.0.0.1:11434`, reçoit la réponse de `llama3.2:3b`, puis conserve l'échange dans la session. La fonction d'appel choisit le mode réel ou le mock ; la configuration ne doit jamais devenir une donnée de session exportée.

La frontière importante est celle entre le serveur Ollama et l'interface : si `ollama list` fonctionne mais que l'application reçoit une erreur de connexion, le problème se situe généralement sur l'URL, le port ou le processus `ollama serve`. Si la connexion fonctionne mais que le modèle est absent, `ollama pull llama3.2:3b` corrige le prérequis. Le mock permet de vérifier le reste sans confondre ces diagnostics.

1. Préparer Python et Ollama

Après extraction des sources, vérifiez que le module `llm_client` est bien disponible avec le projet : il provient du code fourni, pas de l'installation de Streamlit. Le terminal qui lance l'application doit utiliser le même environnement Python que celui qui a reçu les dépendances. Une erreur d'import à ce stade concerne l'assemblage du projet, avant tout appel au modèle.

Le dossier `src/` accueille l'application et `outputs/` les exports de démonstration. Activez l'environnement virtuel dans le terminal qui lancera Streamlit. Ollama, lui, s'installe au niveau du système et sert le modèle sur le port `11434` ; il n'a pas besoin d'être installé dans l'environnement Python.

Sur Linux, installez Ollama avec la commande officielle suivante. Sur macOS et Windows, utilisez l'installateur proposé par la page officielle Ollama, puis ouvrez l'application une première fois.

Code de l'étape

Terminal · Bash

```
mkdir -p assistant-streamlit-local/{src,outputs}
cd assistant-streamlit-local
python3 -m venv .venv && source .venv/bin/activate
python -m pip install streamlit
curl -fsSL https://ollama.com/install.sh | sh
```

Si Ollama est déjà installé, ne relancez pas l'installation : contrôlez simplement sa version et démarrez le serveur si l'application de bureau ne l'a pas fait automatiquement.

Télécharger et vérifier le modèle

`ollama pull` télécharge les poids du modèle une seule fois dans le stockage local d'Ollama. La taille et le temps dépendent de la machine ; gardez suffisamment de mémoire vive et d'espace disque avant de lancer la commande.

Terminal · Bash

```
ollama --version
ollama serve
ollama pull llama3.2:3b
ollama list
```

La commande `ollama serve` reste active dans ce terminal. Si Ollama est déjà lancé comme service, vous pouvez laisser ce terminal fermé et conserver uniquement le processus existant.

Sortie obtenue

Sortie / exemple

```
Successfully installed streamlit
llama3.2:3b    3B    2.0 GB
```

Commande de vérification

Terminal · Bash

```
streamlit --version
curl http://127.0.0.1:11434/api/tags
```

La réponse JSON de `/api/tags` doit contenir `llama3.2:3b`. Ce contrôle vérifie le serveur et le téléchargement du modèle avant d'impliquer Streamlit.

2. Connecter Streamlit à Ollama

Ollama propose un endpoint local compatible avec le format de messages OpenAI. Le client du projet utilise `urllib` de la bibliothèque standard : aucune dépendance OpenAI n'est nécessaire. `GENAI_BASE_URL` contient l'URL du serveur, `GENAI_MODEL` le nom exact retourné par `ollama list`, et `GENAI_API_KEY` reste facultative pour Ollama local.

La fonction `answer` fournit le même contrat au reste de l'application dans les deux modes : un objet avec une réponse et une indication de backend. En mode réel, la réponse provient du modèle local ; en mock, la valeur est déterministe et sert à tester le comportement de l'interface, pas la compréhension d'une question.

Le libellé `Ollama · llama3.2:3b` désigne l'origine technique de la génération ; il ne constitue pas une citation documentaire. Si vous voulez afficher une preuve, il faudra ajouter une vraie source au contrat de réponse, par exemple un document récupéré dans un corpus. L'export sérialise ensuite l'historique avec `ensure_ascii=False` pour conserver les accents et `indent=2` pour faciliter sa relecture.

Dans `src/llm_client.py`, placez la configuration Ollama derrière des variables d'environnement. Dans `src/app.py`, gardez un mode mock explicite pour le test sans réseau, puis activez la branche réelle avec l'option `--real`.

Code de l'étape

Python

```python
import json
import os
import urllib.request
from pathlib import Path

def chat(messages):
    base = os.getenv("GENAI_BASE_URL", "http://127.0.0.1:11434/v1").rstrip("/")
    model = os.getenv("GENAI_MODEL", "llama3.2:3b")
    payload = json.dumps({
        "model": model,
        "messages": messages,
        "temperature": 0,
    }).encode()
    request = urllib.request.Request(
        f"{base}/chat/completions",
        data=payload,
        headers={"Content-Type": "application/json"},
    )
    with urllib.request.urlopen(request, timeout=120) as response:
        body = json.loads(response.read())
    return body["choices"][0]["message"]["content"]

def answer(question, mock=True):
```

```
    if mock:
        return {"answer": "Réponse locale de test.", "source": "fixture.md"}
    model = os.getenv("GENAI_MODEL", "llama3.2:3b")
    return {
        "answer": chat([{"role": "user", "content": question}]),
        "source": f"Ollama · {model}",
    }

def export_session(history):
    Path("outputs").mkdir(exist_ok=True)
    payload = {"messages": history}
    Path("outputs/session.json").write_text(
        json.dumps(payload, ensure_ascii=False, indent=2), encoding="utf-8"
    )
    return payload
```

Configurez ensuite l'endpoint local. Ces variables ne sont pas écrites dans `session.json` :

Terminal · Bash

```
export GENAI_BASE_URL="http://127.0.0.1:11434/v1"
export GENAI_MODEL="llama3.2:3b"
```

Sur PowerShell, utilisez l'équivalent suivant :

powershell

```
$env:GENAI_BASE_URL = "http://127.0.0.1:11434/v1"
$env:GENAI_MODEL = "llama3.2:3b"
```

Sortie obtenue

Le serveur local est joignable et le client connaît le modèle sélectionné.

Sortie / exemple

```
ollama=http://127.0.0.1:11434/v1 model=llama3.2:3b
mock_self_test=ok
```

Commande de vérification

Terminal · Bash

```
python src/app.py --self-test
```

Ce contrôle utilise volontairement le mock pour ne pas consommer le modèle. La vérification réelle aura lieu après le lancement de Streamlit avec `--real`.

3. Construire l'interface Streamlit

Le script est réexécuté lors des interactions avec l'interface ; les messages à conserver doivent donc rester dans `st.session_state`, et non dans une variable locale seulement. Essayez une question normale, puis une saisie constituée d'espaces : le contrôle avec `strip()` doit empêcher cette dernière d'ajouter un échange. Vérifiez l'état de la session autant que le message visible à l'écran.

Streamlit réexécute le script lors des interactions. Une liste créée comme simple variable locale serait donc réinitialisée ; `st.session_state.history` permet de conserver les échanges au cours de la session. L'initialisation conditionnelle évite d'écraser l'historique à chaque clic.

Suivez le branchement du bouton « Répondre » : une saisie vide provoque un message local, tandis qu'une question valide appelle `answer` et ajoute deux messages. La variable `real_mode` permet de garder le même écran pour Ollama et pour le mock. Le bouton d'export transforme l'historique courant en fichier téléchargeable ; essayez-le après plusieurs questions pour vérifier que les échanges ont bien été conservés.

Ajoutez un titre, un champ texte, les boutons « Répondre » et « Exporter », puis un message d'erreur pour une question vide. Les messages sont rendus avec `st.chat_message` afin de conserver une lecture claire de l'échange. L'aperçu « Interface Streamlit » dans la galerie des figures reprend cet écran avec le mode réel Ollama ; l'ordre des contrôles correspond au code exécuté. Affichez le backend dans une zone distincte : cette information aide à distinguer Ollama d'une fixture, mais ne doit pas être confondue avec une source documentaire.

Code de l'étape

Python

```python
import streamlit as st

st.title("Assistant GenAI local")
backend_label = (
    f"Ollama · {model_name()} · mode réel"
    if args.real
    else "Fixture locale · mode mock"
)
st.caption(backend_label)

real_mode = args.real
if "history" not in st.session_state:
    st.session_state.history = []
```

```
question = st.text_input(
    "Question",
    placeholder="Ex. Comment contrôler la qualité des données ?",
)
respond_clicked, export_clicked = st.columns(2)
with respond_clicked:
    answer_clicked = st.button("Répondre", type="primary", use_container_width=True)
with export_clicked:
    export_clicked = st.button("Exporter", use_container_width=True)

if answer_clicked:
    if not question.strip():
        st.error("La question ne peut pas être vide.")
    else:
        try:
            result = answer(question, mock=not real_mode)
        except RuntimeError as error:
            st.error(str(error))
        else:
            st.session_state.history.extend([
                {"role": "user", "content": question},
                {"role": "assistant", **result},
            ])

for message in st.session_state.history:
    if message["role"] == "user":
        with st.chat_message("user"):
            st.write(message["content"])
    else:
        with st.chat_message("assistant"):
            st.write(message["answer"])
            st.caption(f"Backend : {message['source']}")

if export_clicked:
    payload = export_session(st.session_state.history)
    st.download_button("Télécharger session.json",
                       json.dumps(payload, ensure_ascii=False, indent=2),
                       "session.json", "application/json")
```

Sortie obtenue

Une question produit une réponse visible et une source ; une question vide est bloquée côté interface.

Sortie / exemple

```
question="Comment contrôler les données ?" answer_visible=true sources=1
question="" validation="question obligatoire"
```

Commande de vérification

Terminal · Bash

```
streamlit run src/app.py --server.headless true -- --real
```

Dans cette commande, `--server.headless true` est traité par Streamlit et `--real` est transmis à `src/app.py`. Le premier terminal doit conserver Ollama actif. Ouvrez ensuite l'URL affichée, généralement `http://localhost:8501`, et posez une question courte comme : « Résume le rôle d'un test automatisé en deux phrases. »

Sortie réelle attendue

La formulation exacte dépend du modèle et de la température, mais l'interface doit afficher une réponse non vide et le backend Ollama :

Sortie / exemple

```
question="Résume le rôle d'un test automatisé en deux phrases."
answer_visible=true
source="Ollama · llama3.2:3b"
session_messages=2
```

Si vous modifiez `GENAI_MODEL`, le nom affiché doit correspondre à un modèle présent dans `ollama list`. Une réponse générée n'est pas une preuve : pour obtenir des citations, il faudra connecter un corpus ou un pipeline RAG.

4. Vérifier la réponse Ollama et la session

Avant d'auditer la session, vérifiez les trois couches du flux : `curl` confirme qu'Ollama répond, Streamlit confirme que la question atteint `answer`, puis `session.json` confirme que l'échange est conservé. Cette lecture par couches évite de corriger le code d'interface lorsque le vrai problème est un modèle non téléchargé.

Distinguez ensuite le fichier écrit côté serveur et celui téléchargé par le navigateur : l'export doit contenir les mêmes messages, dans le même ordre. Effectuez au moins un échange avant de tester un contrôle qui exige une session non vide. Relisez le JSON téléchargé avec un éditeur et vérifiez que seules les informations nécessaires à la conversation y figurent, sans paramètres d'accès au service.

L'export comporte deux destinations : le fichier écrit dans `outputs/` sur la machine qui exécute Streamlit et le téléchargement proposé au navigateur. Comparez leur contenu pour vérifier qu'ils représentent la même session et que les champs conservés correspondent aux échanges affichés.

Le passage dans `json.tool` contrôle la syntaxe du fichier. La recherche de marqueurs comme `API_KEY` ou `TOKEN` constitue un contrôle complémentaire très simple : elle ne détecte pas tous les secrets possibles et peut signaler un mot présent dans une question. Examinez donc les champs exportés et vérifiez surtout que le programme n'y ajoute jamais ses variables de configuration.

Testez le bouton d'export, ouvrez le JSON et vérifiez qu'il ne contient ni clé, ni variable d'environnement, ni cookie. Le modèle reste local, mais une question et sa réponse peuvent tout de même contenir des données sensibles : ne partagez pas l'export sans l'examiner.

Sortie obtenue

La session exportée est lisible et ne contient que les données nécessaires à l'audit.

Sortie / exemple

```
session_messages=2 secrets_found=0 export=outputs/session.json
```

Commande de vérification

Terminal · Bash

```
curl -s http://127.0.0.1:11434/api/tags
python -m json.tool outputs/session.json >/dev/null
! grep -E 'sk-|API_KEY|TOKEN' outputs/session.json
```

Code de l'étape

Le test de non-divulgation peut être automatisé avant chaque export :

Python

```python
import json
from pathlib import Path

session = json.loads(Path("outputs/session.json").read_text(encoding="utf-8"))
serialized = json.dumps(session, ensure_ascii=False)
for forbidden in ("sk-", "API_KEY", "TOKEN"):
    assert forbidden not in serialized
assert session["messages"]
print("session sans secret")
```

Exemples de sorties

L'application démarre en local, interroge Ollama, affiche le modèle utilisé et exporte une session sans secret.

Dépannage et diagnostics

Symptôme	Cause probable	Vérification ou correction
`Connection refused` sur `11434`	Ollama n'est pas démarré	lancer `ollama serve` ou l'application Ollama
modèle introuvable	le modèle n'a pas été téléchargé	exécuter `ollama pull llama3.2:3b` puis vérifier `ollama list`
réponse très lente	modèle trop lourd ou mémoire limitée	essayer un modèle plus compact et fermer les applications inutiles
`KeyError: choices`	endpoint ou réponse non compatible	vérifier `GENAI_BASE_URL` et tester `/api/tags`
l'historique disparaît	état stocké dans une variable locale	conserver les messages dans `st.session_state.history`

N'exposez pas le port Ollama à Internet : l'API locale n'est pas une couche d'authentification. Si l'application doit sortir du poste de développement, ajoutez un proxy authentifié, des limites de taille et une journalisation maîtrisée.

Vérification finale

- [ ] Ollama répond sur `127.0.0.1:11434` ;
- [ ] `llama3.2:3b` est présent dans `ollama list` ;
- [ ] réponse réelle obtenue avec `--real` ;
- [ ] mode mock opérationnel pour les tests rapides ;
- [ ] question vide refusée ;
- [ ] modèle Ollama affiché comme backend ;
- [ ] export sans secret.

Limites et prolongements

Ollama garantit l'exécution locale, pas la qualité des réponses. Le modèle `llama3.2:3b` peut manquer de connaissances ou produire des affirmations incorrectes ; ajoutez un jeu d'évaluation, des sources vérifiables et une politique de données avant un usage métier. Si l'interface sort du poste local, ajoutez également une authentification et un contrôle des ressources.

Références

- Streamlit — documentation officielle
- 12-factor app — configuration
- Ollama — téléchargement et modèles locaux
- Ollama — API compatible OpenAI